Synthesising and Verifying Multi-Core Parallelism in Categories of Nested Code Graphs

Christopher Kumar Anand[*]

anandc@mcmaster.ca

Wolfram Kahl[*]

kahl@cas.mcmaster.ca

Software Quality Research Laboratory
Department of Computing and Software, McMaster University
Hamilton, Ontario, Canada L8S 4K1

24 January 2008

Abstract

We present the Multi-Core layer of the larger Coconut project to support high-performance, high-assurance scientific computation. Programs are represented by nested code graphs, using domain specific languages.

At the Multi-Core level, the language is very restricted, in order to restrict control flow to non-branching, synchronising control flow, which allows us to treat multi-core parallelism in essentially the same way as instruction-level parallelism for pipelined multi-issue processors. The resulting schedule is then presented as a "locally sequential program", which, like high-quality conventional assembly code in the single-core case, is arranged for hiding latencies at execution time so that peak performance can be reached, and can also be understood by programmers. We present an efficient, incremental algorithm capable of verifying the soundness of the communication aspects of such programs.

Software Quality Research Laboratory

McMaster University

SQRL Report No. 50

Contents

1 Introduction

Current trends in high-performance processor technology imply that the programming interface is becoming more complicated. The pronounced shift towards pervasive Single Instruction Multiple Data (SIMD) parallelism, and the introduction of heterogeneous computing elements, including hierarchically distributed memory (encompassing conventional distributed memory, streaming memory, software-directed caches, and private memories), will require changes in software development tools.

The Cell Broadband Engine is the most widely deployed in the first generation of such processors. Using its resources efficiently requires explicit synchronisation of both code and data, and packaging of both to fit in small memory spaces and constrained communication infrastructures.

The Coconut (COde CONstructing User Tool) project has the aim of capturing the entire design and development process for high-performance scientific software, with a focus on image and signal processing, in a sequence of formal specifications. The hardware-specific parts of Coconut currently target a single Cell processor with two logical PowerPC threads and eight physical SIMD compute engines, each with its own private memory.

The Coconut tool set uses as intermediate representations of programs and program fragments a variant of term hypergraphs we call "code graphs".

Term graphs have a long history as an efficient implementation of symbolic expressions. A certain kind of term graphs also provides a natural model of a "resource-conscious" kind of categories (gs-monoidal categories, [8]); this provides a useful justification of many kinds of algebraic data-flow graph transformations. To combine control flow and data flow, we nest the simple data-flow code graphs inside an outer-level code graph with control-flow semantics, similar to the two different semantics of the flownomials of [28]. Software pipelining, i.e., instruction reordering to make better use of instruction pipelines and multiple-issue queues, is an algebraic transformation involving both levels of such nested code graphs.

This approach naturally extends to modelling (restricted) concurrency: We introduce a third, outer layer for explicit concurrency, with a semantics more on the data-flow side (this is related with the implicit concurrency in pure functional programs), but again with some control-flow characteristics.

Algebraic transformations crossing the outer two levels in such three-level nested hypergraphs are used to map potential concurrency to explicit concurrency, and to hardware parallelism. On the level of multi-core processors this will involve the automatic introduction of appropriate synchronisation instructions, or, in the opposite direction, the elimination of such concurrency primitives for the purpose of verifying against a concurrency-independent specification.

This paper presents the control flow at the middle "multi-core" level. In conventional programs, this level of control flow is usually encoded in library calls (*e.g.*, MPI [23, 27]) in the program text. Our approach is to make this disjoint in a separate level of the graph. This allows for a simple verification of correctness for deeply pipelined data flow graphs through a network of cores (processors). As a result, whether multi-core parallelism is directed by user code, user code generators, or automatic parallelisation, the programmer/compiler writer can focus on tuning for performance, and check correctness in a computation which grows linearly with program size, and quadratically in the complexity of the hardware.

The distinguishing feature of code graphs at this level is that they are presented as ordered lists of instructions. Proofs of sound parallelisation work by going through the list in order, maintaining a limited set of state information (status of signalling and data transfers, contents of buffers). Any ambiguity (from a race condition) or deadlock can be flagged in the presented program.

Related Work

This work is the first step in bringing together disconnected research efforts in (1) verification of concurrent programs, (2) scheduling for distributed processors, and (3) decomposition of large numerical problems.

Practical parallel computation is a special subset of the set of concurrent programs. We have defined a precise subset of the vaguely defined set "parallel computation". Verification of correctness in this set *should* be simpler than in the larger set, but many important ideas come from efforts to verify general concurrent programs. We make fundamental use of partial orders in analysing concurrency within our programs. Godefroid and several collaborators have demonstrated that partial order methods can be much more efficient than other methods for general concurrent problems. In the language of [13], we construct a partial order in such a way as to show that there are no back-tracking points, i.e., no points where a different execution sequence would have lead to a different result. Our implementation is much simpler, with tight complexity and memory bounds, because it is tailored to the restricted set of programs we consider. The trade-off is that the programs we consider are orders of magnitude larger than typical concurrency benchmarks.

We target the Cell BE processor, using matrix multiplication as our first benchmark, as do [5]. Our approaches differ in that we present the programmer with an unconventional interface via DSLs embedded in Haskell, we schedule parallelism at compile time, and we aggressively pipeline communication and processing. As a result of the aggressive pipelining, correctness is difficult to infer from the program/scheduler text, and we offer a mechanism to verify correctness of the generated program.

There is a lot of literature related to loop decomposition, and specifically the decomposition of large operations in linear algebra. The generation of *consumers* used in our scheduling algorithm implicitly encodes a loop tiling (see [16, 17]). Research in this direction is currently considering much more complex computational examples than we are able to at this time, and will provide a rich set of benchmarks for the future. That said, some of the work on tiling will have to be reconsidered in light of the shift from computational to main memory bottlenecks. We hope that our tool will accelerate the development of patterns of communication involving many point-to-point pipelined transfers.

Overview

We explain in Sect. 2, how directed hypergraph syntax naturally integrates data flow and concurrent control flow, and summarise how we use these *code graphs* in nested arrangements for for different aspects of the generation of high-performance code.

We then turn our attention to implementation aspects of multi-core parallelism (Sect. 3), contrasting it with other kinds of parallelism, to provide the background for the decision (Sect. 4) to define a virtual machine for encapsulating certain low-level aspects of targeting multi-core architectures. The details of this virtual machine are then presented in Sect. 5, and in Sect. 6 we show how we produce, partially guided by graph transformations, a multi-core program scheduled to hide communication latencies as far as possible. To be able to independently verify and certify the concurrency aspects of the result, we introduce, in Sect. 7, an efficient algorithm that checks satisfaction of the relevant safety and liveness properties in a single pass over the "fuzzy" multi-core schedule.

2   Nested Code Graphs

Graphs are frequently used as internal presentations for code generation and analysis purposes; we manage to use a uniform graph syntax at different levels to combine data-flow aspects, control-flow

aspects, and concurrency aspects in nested *code graphs.*

Term graphs, the abstract version of data-flow graphs, are usually represented by graphs where nodes are labelled with function symbols and edges connect function calls with their arguments [26]. We use the dual approach of directed hypergraphs where nodes are only labelled with type information (if applicable), function names are hyperedge labels, and each hyperedge has a sequence of input tentacles and a sequence of output tentacles (each incident with a node):

Definition 2.1 *A hypergraph* $H = (\mathcal{N}, \mathcal{E}, \mathsf{src}, \mathsf{trg}, \mathsf{nLab}, \mathsf{eLab})$ *over a node label set* NLab *and an edge label set* ELab *consists of*

- *a set* $\mathcal{N}$ *of* nodes *and a set* $\mathcal{E}$ *of* hyperedges *(or* edges*),*
- *two functions* $\mathsf{src}, \mathsf{trg} : \mathcal{E} \to \mathcal{N}^*$ *assigning each hyperedge the sequence of its* source nodes *and* target nodes *respectively, and*
- *two functions* $\mathsf{nLab} : \mathcal{N} \to \mathsf{NLab}$ *and* $\mathsf{eLab} : \mathcal{E} \to \mathsf{ELab}$ *assigning labels to nodes and hyperedges.*

Hypergraph homorphisms are pairs consisting of a total node mapping and a total edge mapping, preserving src, trg, nLab, and eLab, and induce a category.

If a hypergraph in some sense represents a program, it has an input/output interface:

Definition 2.2 *A* code graph $G = (H, \mathsf{In}, \mathsf{Out})$ *over a node label set* NLab *and an edge label set* ELab *consists of*

- *a hypergraph* $H = (\mathcal{N}, \mathcal{E}, \mathsf{src}, \mathsf{trg}, \mathsf{nLab}, \mathsf{eLab})$ *over* NLab *and* ELab*, and*
- *two node sequences* $\mathsf{In}, \mathsf{Out} : \mathcal{N}^*$ *containing the* input nodes *and* output nodes *of the code graph.*

A code graph homomorphism is a hypergraph homomorphism that additionally preserves In and Out.

Typing aspects are dealt with in the usual way:

Definition 2.3 *A hypergraph signature is a hypergraph where the labelling functions* nLab *and* eLab *are the identities on the respective label sets.*

A typed hypergraph *is a hypergraph together with a homomorphism into a hypergraph signature.*

The resulting categories of (typed) hypergraphs have pushouts, and can therefore be used as basis for the double-pushout approach to graph transformation [9], where rules are induced by spans in the category of code graphs [2].

Code graphs also give rise to a gs-monoidal category with sequences of node labels as objects and code graphs as morphisms; [8] use this to establish functorial semantics for code graphs, and [10, 20] show examples where this semantics uses locally ordered categories, such that code graph hmomorphisms $F : G_1 \to G_2$ document ordering relations $\llbracket G_2 \rrbracket \sqsubseteq \llbracket G_1 \rrbracket$ on the semantics.

2.1   Data Flow Graphs and Parallelism

Using code graphs as term graphs follows the "jungle" view of term graphs, initially put forward by Hoffmann and Plump [19, 24], and equipped with functorial semantics by Corradini and Gadducci [8]. In such code graphs, nodes are labelled with types and in a certain sense represent values of the respective types; hyperedges are labelled with operations taking input values as arguments and producing results as outputs.

Data flow semantics of hypergraphs implies the following:

- If an edge has multiple input tentacles, the computation associated with it has multiple argument positions, one for each input tentacle.

  This is the only kind of branching present in term trees. This kind of branching is also the source of potential parallelism, particularly in functional programming, since it represents a split of the computation of the outputs of this edge into potentially independent computation of its inputs.

- If a node is attached to multiple input tentacles, the value represented by this node is used by several consumers.

  This *sharing* marks the transition from term trees to term graphs, and is frequently only a tool for efficiency (shared effort) without semantic relevance, for example in most implementations of functional programming. (Sharing does have semantic relevance for example in functional-logic programming [1] and in multi-algebras [10, 11].)

- If an edge has multiple output tentacles, the computation associated with it produces multiple results, one for each output tentacle.

  This presents a relatively unproblematic generalisation of term trees or term graphs.

- The possibility that a node is attached to multiple *output* tentacles was used in [20] to represent "joins" in data flow, i.e., different equivalent ways to produce a value.

Cycles in term graphs are frequently used to encode recursions; this normally uses a least fixed-point semantics and relies on the presence of non-strict constructs, in particular conditionals or, in lazy functional programming, datatype constructors.

In the central assembly-level components of Coconut, assembly op-codes (which all have strict semantics) serve as edge labels, with inputs corresponding to argument registers and outputs corresponding to result registers. We use acyclic data-flow code graphs with these op-codes as edge labels to represent loop bodies of scientific computation applications, and use graph transformation and analysis to exploit instruction-level parallelism and achieve optimal schedules [3, 2].

2.2   Non-Concurrent Control Flow Graphs

The archetypal control flow graphs are finite state machines; nodes are labelled with state types and represent states, and edges labelled with actions, sometimes perceived as active (operations), sometimes as passive (recognition of input symbols), and sometimes as both (e.g. in Mealy machines).

For conventional edges, finite-state machines serve as the standard control flow model, and represent non-concurrent control flow.

Cycles in control flow graph express iteration, and correspond to the asterate operator of Kleene algebras.

At the lowest (output) level in Coconut, assembly *instructions* (except conditional branches), i.e., op-codes together with register references, serve as control flow edge labels, with the "state before" as single input, and the "state after" as single output.

The conversion of op-code-labelled hyperedges to instruction-labelled control-flow edges happens as part of *scheduling* of the data-flow graphs; since scheduling essentially means adding of control dependencies, and scheduling of a data-flow graph together with a register allocation enforces the data-flow dependencies via the stronger control-flow dependencies.

At a higher level, we represent computational kernels as sequential control-flow graphs where most of the edges are labelled with larger data-flow graphs, and implement software pipelining essentially as graph transformation on these control-flow graphs [4].

2.3   Control Flow Graphs and Concurrency

The standard model for *concurrent* control flow are Petri nets, which are, in more theoretical work (e.g. [14]), formalised as directed hypergraphs, with places represented by hypergraph nodes and transitions by hyperedges (but with *multisets* of input and output places for each transition — Sassone introduced the concept of "pre-net" which has sequences instead of multisets [25, 6]).

Putting everything together, control flow semantics of directed hypergraphs therefore implies the following:

- If a node is attached to multiple input tentacles, it is normally understood that control propagates along any applicable outgoing edge.

  This kind of branching implements decisions by making different edges applicable depending on the circumstances. In finite-state automata recognising regular languages, those edges the label of which matches the next available input are applicable; more generally, conditions depending on the "current state" can be used. If several outgoing edges of the same node are applicable, then this kind of branching introduces non-determinism.

  If this is the only kind of branching, the resulting graphs can be used as decision trees, or as tree structures for branching-time temporal logics [12].

- If a node is attached to multiple output tentacles, then the corresponding different operations pass control to the same state.

  This corresponds to moving from tree structures for branching-time temporal logics to graph structures, and similarly from infinite regular languages to finite automata.

- If an edge has multiple output tentacles, we understand this as forking the current thread of control into multiple threads of control.

  This corresponds to the fact that Petri net transitions with multiple outgoing edges place tokens into each successor place upon firing.

- If an edge has multiple input tentacles, we understand this as introducing synchronisation of concurrent threads.

With this background, it becomes clear that, in the non-concurrent case. conditionally branching assembly instructions should not be represented by branching hyperedges, but instead as two conventional edges of which in each state only one is applicable.

2.4   Concurrent Interpretation of Data-Flow Graphs

From the theoretical literature [14, 25, 6] one notices that Petri net *computations*, which are finite acyclic versions of Petri nets equipped with a start and end interface, give rise to essentially the same kinds of categories as term graphs, namely variants of gs-monoidal categories [8].

The description in the previous section was accordingly tuned to emphasise the compatibility between the concurrent control-flow view of a hypergraph with the data-flow view, as long as no cycles are present.

Indeed, Sassone's pre-nets [6], designed to reflect the "individual-token philosophy" to composition of Petri net computation can be understood as making individual threads of control explicit.

The two views combine easily into data-flow graphs with concurrency aspects, where the distribution aspects are most easily made explicit by adding "host identifiers" to nodes and hyperedges. For an example see Fig. 3.

Combining this representation of concurrency aspects of distribution with the nested hypergraph representation of sequential computational kernels, we obtain a nesting of hypergraphs with at least three three-levels — implementing iteration of concurrent behaviour will require wrapping the concurrent control-flow code graph at the third level into a cyclic sequential control-flow graph which then produces a four-level nesting.

3   Efficient Multi-Core Parallelism

Different levels of parallelism raise different issues with respect to efficiency and correctness. It is useful to put Multi-Core Parallelism into a context categorised in order to contrast current issues with parallelism issues in the past.

Distributed parallelism involves execution on multiple hosts, also called "nodes", each running an operating system.

> Long and variable latencies, and network and node failures during computation may be significant issues in this context. The message-passing library MPI [23, 27] is most commonly used for distributed scientific processing, typically implementing a Single Program Multiple Data (SPMD) model. The most feature-rich tools for distributed parallelism are marketed as "grid computing".

> The SPMD approach allows programmers to write a single program which runs on multiple nodes. Each instance therefore contains code for all instances and knowledge of the data distribution across nodes. This model is not suited to heterogeneous multi-core architectures with small private memories.

Instruction-Level Parallelism (ILP): CPUs need multiple execution units for different types of computation to reduce design complexity. Instruction-Level Parallelism (ILP) takes advantage of this hardware design imperative by dispatching instructions to multiple units in parallel. In the vast majority of cases, the CPU has circuitry to introduce this parallelism into a sequential stream of instructions, and may make other modifications (e.g., register renaming) to improve throughput. Using this mechanism, sequential programmers and basic compilers get a some degree of parallel execution for free, while advanced compilers can produce significant levels of parallelism (e.g. four-way on recent Power architectures). Very Long Instruction Word (VLIW) architectures expose this level of parallelism explicitly to the compiler.

Single Instruction Multiple Data (SIMD) properly refers to architectures with hardware support for large-vector arithmetic operations. These were among the most successful early "super computers".

Vector-in-register parallelism (VIRP) In current architecture, SIMD most often refers to short-vector in a register architectures designed to accelerate the manipulation of digital media. Typical instructions treat a 128 bit instruction as four 32-bit integers and perform operations on them in parallel. Although some early implementations could accurately be described as SIMD, current instruction sets have added rich "horizontal" instructions which operate across elements in a single register values, for example, by shuffling bytes within a single register value [7].

Multi-core architectures are evolving rapidly, nevertheless, we can assume the following characteristics: computational resources are organised into units similar to conventional CPUs, with asynchronous execution in the multiple cores, and weakly coherent access to common memory resources. The programming model for these resources differs greatly: private memory, user-directed caching of global memory, or hardware-directed caching of global memory. The models range from highest potential performance, to simplest programming model.

ILP has been so successful that almost no programmer considers it when writing code. Large-vector SIMD is too restricted to apply to a wide enough range of applications to make it viable. Distributed parallelism always requires significant programmer intervention, and a whole industry has grown up around providing tools to write and debug distributed parallel programs. We have previously shown that taking advantage of VIRP requires the development of new patterns of efficient execution, and that an approach marrying a DSL programmer interface to a system based on graph transformation can produce highly efficient, highly readable, and high assurance code [3].

Our goal is to duplicate the success of ILP and VIRP at the multi-core level, avoiding as many of the problems associated with distributed parallelism as possible. Current SPMD models for distributed parallelism raise the cost of developing software above what most applications can support. Run-time requirements for library and operating system support do not scale well to large numbers of cores, and exceed the capabilities of some light-weight cores.

Why is ILP universally exploited while SPMD is considered an arcane art? Accepting that it makes sense to compare such different levels of parallelism, we believe that the answer lies in the way the two are understood (both formally and informally). SPMD is implemented as a library, while ILP is implemented as a language (machine language). SPMD can be used to express any programming construct and implemented at any scale, and most support libraries try to abstract away the scale and architecture of the underlying hardware. The level of parallelism supported is unbounded. Any conceivable program can be implemented. Unfortunately, most conceivable programs are not correct, and it is difficult in practice to tell which ones.

ILP, on the other hand, is defined in terms of a language (machine or assembly language). Both resources and the amount of parallelism are bounded. All resources whose efficient use determines performance are accessible to the programmer through instruction (re)scheduling. For the most common architectures, the programming model is strictly sequential within the CPU. Some architectures allow external observers to see limited non-sequential execution (by reordering external memory accesses for performance reasons).

Obviously, presenting a sequential model to the programmer makes ILP easier to use. Note, however, that programmers (and more commonly compiler writers) aiming for peak efficiency must be aware of non-sequential (out-of-order) program execution. The key point is that correctness can be determined using the sequential programming model or the data-flow-graph-based model, even if significant changes in scheduling are required to efficiently use resources. Furthermore, the limited resources in support of parallel execution make it possible to characterise the set of schedules and design effective heuristics for searching within it.

4   Motivation for Multi-Core Virtual Machine Model

Our aim is to encapsulate the important features of multi-core parallelism in a virtual machine (language with semantics) with the hope that techniques developed for ILP will apply to parallelism at this level. This is our attempt to find a balance, exposing enough details to enable high-performance computation while hiding enough details to make it easy for the programmer to understand program semantics.

The most important feature we need to capture is data locality. The hardware required to present the programmer with a flat memory model across multiple execution threads is very expensive, and introduces performance-reducing non-determinacy. Processors designed for signal processing have for many years provided user-directed cache modes, including directly addressable caches, in which the programmer can fix part of the cache to memory mapping in order to guarantee fast access to important data structures. Because such optimisations are non-portable and require support from the

operating system, they have not been used in multi-user servers or desktop processors.

The Cell SPUs[18] have no cache, but rather have private local storage. Communication with the rest of the processor must be performed explicitly using direct memory access (DMA) transfers. This can be seen as integrating *on-chip* the architecture of distribution that has existed for decades on the circuit boards of workstations, where microprocessor and graphics processor (GPU) execute independently, and communicate via DMAs.

Data locality poses a challenge for synchronisation for both cached and non-cached memory architectures. This is because access to memory which appears to be local may require distant access. In a cached architecture, this occurs when two physical cores access the same physical memory causing the bus hardware to transfer cache lines between the two processors, introducing delays. To avoid this, our language does not contain access to non-local memory, even as part of inter-process signalling, which implies that we do not use mutexes.

Our virtual machine must contain multiple threads of execution. Each thread has a small list of local resources: signal registers which can be set by remote processes and data buffers to store blocks of data for computation. These are the "registers" of our virtual CPU, and pure, local function computation and asynchronous communication[1] are two execution units. We can synthesise branching control flow inside the pure functions just as we did for VIRP, and we can move it to a higher level in the Coconut code graph. Restricting computation to pure functions, and control flow to thread synchronisation significantly simplifies the process of reasoning about parallelisation for the programmer or compiler and the verifier.

By separating control and data flow, this allows a single-threaded control process which may be distant from some or all of the computation engines to originate the complex data flow patterns necessary in scaling up multi-core computations, while separating algorithm-intrinsic control flow from control flow required by parallel computation. In this way, the algorithm designer can solve one problem at a time: writing a correct algorithm and writing an efficient parallelisation strategy. Unlike ILP and like SPMD, to tune performance, the programmer must explicitly insert parallelisation, although, in Coconut, this is done in program generators, not programs, and can be simplified by using higher-level patterns. Like ILP and unlike SPMD, verification ensures that the program as a whole is a pure function, *i.e* independent of parallel execution order.

5   Virtual Machine Language

A sequence of computations is a list of pairs (c, i) of execution core identifiers and instructions. In the present description, the instructions on each core execute in order, although just as in the ILP case, it would be easy to introduce limited out-of-order execution managed by the run-time system or dedicated hardware.

Data movement instructions, however, are assumed to run asynchronously from other computations, so the exposed instructions only initiate data movement/signalling or wait for it to complete. DMA engines to handle DMA are almost always implemented as autonomous processors executing transfer lists. We embed the data movement with the purely local computations on a core because this is analogous to the interleaving of load/store instructions and register based computation on a Reduced Instruction Set Computer (RISC). Unlike load/store instructions in a RISC CPU, we have to separate the request for transfer from the acknowledgement of completion — in RISC processors, issuing the instruction using the target register of a load instruction will stall the execution pipeline until the data

---

[1]Although single communication primitive would be sufficient, we separate data flow and signals because machine architectures can (and should) optimise one of throughput and the other for latency. On the Cell BE signals are implemented as both signals and messages, and data flow as DMA transfers.

has been transferred, while DMA engines do not provide equivalent synchronisation mechanisms.

We choose to expose this for several reasons: Signalling may compete with data transfers on a common bus, and this allows the instruction scheduler to take this into account. Data transfers themselves introduce ordering, so signalling may not always be necessary. The separation makes it easier to maximise latency hiding for *two-way* communication within given resource constraints. Each transmission executes asynchronously with a variable latency, so it is useful to separate them enough to hide the expected latency.

While computation within a core is in-order, computation on different cores is *a priori* unordered, as are all data transfers. For this to work, the computational threads on the send and receive side must refrain from writing to the source and target areas, and the receive side must refrain from reading from the target area, until completion of the transfer has been signalled.

To take advantage of the higher efficiency of large transfers, signalling of DMA completions is handled via additional data — a "tag" — contained in the transfer. This is the method we use on the Cell BE; it is possible to do this since the sequence of writes to DMA target addresses can be ordered using fences[2]. Detection of a tag (which must be different than the tag associated with the previous buffer contents) indicates to the receiver that the transfer has completed. We will assume that the sender also has a mechanism for detecting completion.[3] And a signalling mechanism, with signal status stored locally (in local private memory, or a special hardware register) on each core.[4]

Cores are referenced by *CoreId*. To each core, we will associate the following state:

state	reference	function	values
signal registers	*SignalId*	enumeration of a small, predetermined number of Boolean values local to each core, but writable from any core	can be *set* or *unset*
buffers	*LocalAddress*	store blocks of data	*readable*, or *writable* by set of *CoreId*s, or exclusively for *local* access
data tags	*DataTag*	tags associated with data buffers	contain integers

Two examples are helpful at this point. In Fig. 1, we introduce a graphical representation of the virtual machine language. Following UML convention, communication is asymmetric, so it is shown with half arrowheads. There is both synchronisation and data transmission, so thin and thick lines

---

[2] DMA engines usually support lists of blocks to transfer. Transfer out of order may be more efficient, so order is usually ignored, but different constructions (bit flags on the DMA list instructions) may be available to enforce ordering within a particular list of blocks, and within all lists of blocks queued up for transfer. By putting the tags in a separate atomic unit for transfers (aligned 16-bytes), and putting this line in a separate DMA block, following a fence, following the list of blocks needed for the data transfer, the tag will not appear to the receiver until after all data has been transferred.

[3] On the Cell BE processor, DMA engines are paired with computational cores, and detecting DMA completion is low-cost and does not involve bus traffic. This is the preferred setup from a hardware point of view, because each private local memory needs an interface to the on-chip network fabric, and this requires that most of the functionality of a DMA engine is implemented in any case.

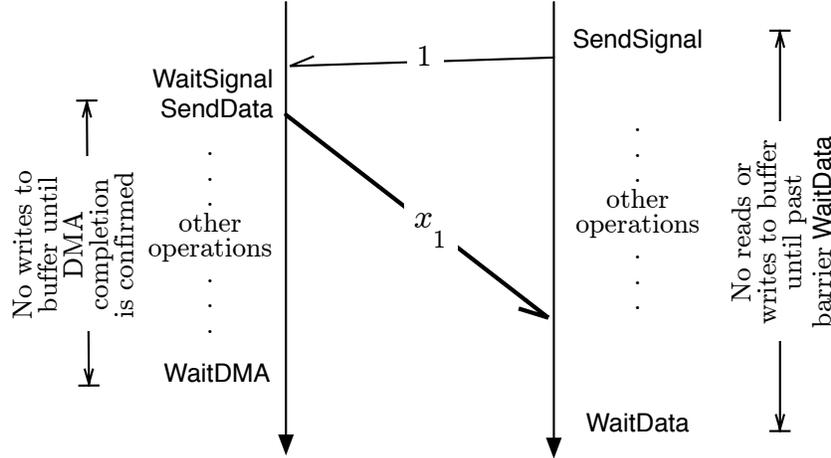[4] On the Cell BE, we use the SPU signal registers.

Figure 1: Simple data transfer requires one "ready" signal before data is transferred.

are used to represent the respective low and high bandwidth requirements. Signals are labelled by their *SignalId*, and data is labelled by a symbol representing the data being transferred, subscripted by the data tag.

In a simple transfer, the receiving core sends a pre-assigned signal to indicate that the receiving buffer is available for writing. It is up to the scheduler to ensure that no local or remote reads or writes to this buffer occur. In the case of local reads and writes, this is easy to verify visually, by checking the inputs and outputs of all pure functions between the SendSignal and WaitSignal. Ensuring that no DMA activity sources or targets this buffer during this time is more difficult, and requires the establishment of a partial order on the instructions executing on different cores. Similarly, the scheduler must ensure that the source buffer is not written to between the WaitSignal and WaitDMA instructions, although reading is allowed, and is effective in pipelining instructions.

In the more complex example shown in Fig. 2, a single signal is sent to indicate that the destinations for a whole chain of transfers are available. This amortises the cost of the signalling over multiple transfers, at the expense of potentially tying up buffers for longer periods of time. It may be easier for the scheduler to software pipeline than a sequence of simple transfers, because there are fewer latencies to worry about. For algorithms requiring numerous transfers of small amounts of data, the reduction in signalling bandwidth may also be significant. In this example, we introduce conventional mathematical notation for pure functions, f, g, and h. Since the data transfers in one cycle have distinct target cores, a single data tag can be used for all transfers. To achieve good performance, the signals and data transfers would be software pipelined within each node, so that, for example, in a simple mapping of $h \circ g \circ f$ over an array $\{x_i\}$, with minimal pipelining, the evaluation of $f(x_{i+2})$ would overlap the evaluation of $g(f(x_{i+1}))$ and the calculation of $h(g(f(x_i)))$.

Virtual Machine Instruction List

We present the syntax of the virtual machine instructions, together with high-level description of their intended effects. We also summarise how we implement these for the current Cell BE architecture, how they could be implemented on other architectures, and their performance implications.

There are no nested instructions. Each instruction takes a sequence of operands from the finite types *CoreId*, *SignalId*, *DataTag*, *LocalAddress*, *GlobalAddress*. The sizes of these types are determined by the specific hardware/software architecture.
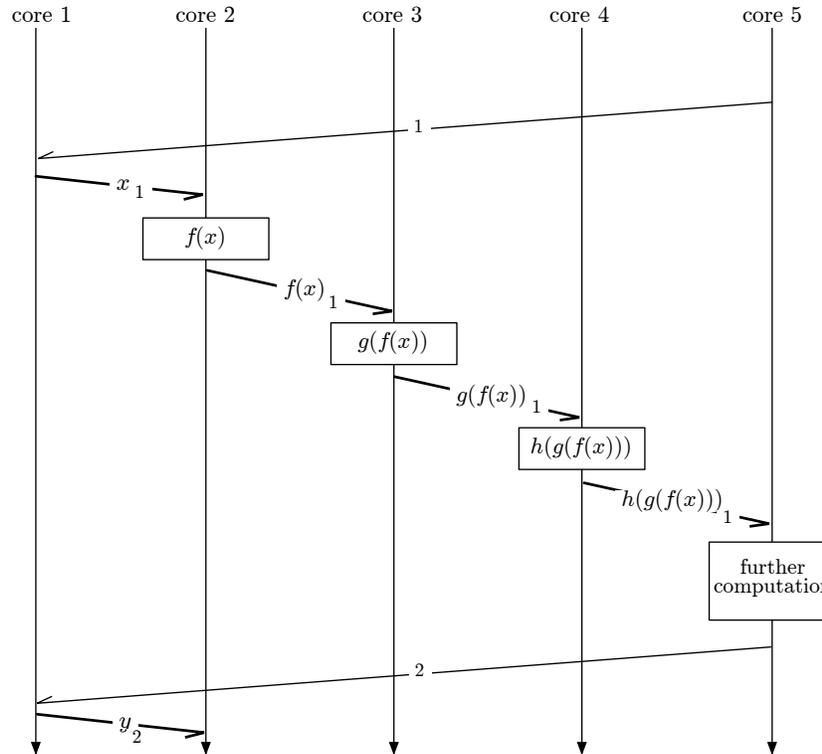
SendSignal   *CoreId SignalId*

Figure 2: Complex transfer, minimising signal traffic.

Send the specified signal to the specified Core. If the signal is already set, it remains set. The receiver must unset it (with a WaitSignal) in order to be able to detect the next signal. Any core can send any signal to any other core. It is up to the programmer/generator to assign signals just as registers are assigned so that different uses do not overlap.

This command is currently used to issue a "clear to send" notice so that a remote SPU is able to initiate a DMA data transfer. When used as a "clear to send" notice, the buffer to which the requested data will be written should have already been marked as free with the FreeBuffer command. If the buffer is not so marked, it will be impossible to detect locally when the buffer is filled with new data.

WaitSignal *SignalId*

Pause execution on this core until the specified signal has been received.

Efficiency of the schedule depends on this instruction only occurring in the execution list for a core after the signal has already been received by the underlying hardware. In this case the latency of the transfer is completely hidden.

SendData *LocalAddress DataLength GlobalAddress DataTag DMATag*

Initiate a DMA transfer to send a data block from the memory of this core to the private memory of a second core, using its address in the global address space.

In the current implementation, the *DataTag* is written to a reserved memory word preceding the buffer, and the data transmission is adjusted to include it in the transmission.

The *DMATag* must be used on the same core with a WaitDMA instruction to verify the end of the transfer before local writes are allowed. On the Cell BE, it is possible to interrupt on DMA completion, which can be used to reduce the latency between initiation of a send and the next use of the buffer. This introduces interrupt overhead into the run-time system, and significantly complicates the verification of the run-time system.

WaitDMA   *DMATag*

> Wait for completion of the DMA transfer with the given tag. Only the initiating core can check this tag. Another core, the controller, or off-node I/O must infer that this transfer is complete from other synchronising instructions.

WaitData   *LocalAddress DataTag*

> Wait for the data block located at the given local address to have fully received an inbound transfer identified by the given data tag.

> In the current Cell BE implementation, this is done by repeatedly reading the data tag from the local private memory until the expected tag is read. This is convenient because it is possible to put fences in DMAs, ensuring that the data tag is transferred after the block data is transferred.

> More efficient hardware would support sending a signal after a DMA transfer is complete. To implement this on the current Cell BE, one would have to wait for the local signal for DMA completion and then send a signal. This would introduce either a lot of extra latency, or, if interrupts are used, extra overhead on the sending core for the interrupt handler, and significantly more complicated verification of the run-time system for the cores.

FreeBuffer   *LocalAddress*

> Indicate that the buffer at the given local address is free to be written to.

> This has to occur before an incoming data transfer request is initiated when the previous value of the data tag is not known in order to be able to tell when the data block transfer has completed. For simplicity of generation or verification, each WaitData may be paired with one.

> In the current Cell BE implementation, this instruction causes a byte to be written to a reserved memory location preceding each buffer. If light-weight signals are supported for signalling DMA completion to remote locations, this instruction might be required to reset a hardware register.

LoadMemory   *LocalAddress DataLength GlobalAddress DMATag*

> Initiate a DMA transfer of the specified data from main node memory to the core's private memory. The source buffer can be written to, and the target used after checking the *DMATag*.

StoreMemory   *LocalAddress DataLength GlobalAddress DMATag*

> Initiate a DMA transfer of the specified data from the core private memory to node main memory The source buffer can be written to, and the target used after checking the *DMATag*.

LoadComputation   *Computation GlobalAddress DMATag*

> Load new computational kernel from location in main memory specified in the *Computation* structure.

> This duplicates the functionally of LoadMemory with a different type. Separating them makes verification a lot easier, and takes advantage of the encapsulation of *Computation*.

> In the Cell BE implementation, they are converted to LoadData during production of the byte streams.

RunComputation   *Computation LocalAddress* LocalAddress* Parameter**

> Run the specified computational kernel, using a list of addresses for input and output buffers, and a list of parameters. The signature of the computation includes information on the sizes of the buffers and which input and output buffers may coincide (for in-place computations). This information is needed to evaluate the instruction stream into a data flow graph.

> The computation is a compiled pure function, adhering to a defined binary interface. The binary interface allows a list of input data buffers, a list of output data buffers and a list of parameters

to be passed to the function. In the Cell BE implementation, the pure functions are generated from an source code in an existing DSL.

6 Scheduling Algorithm

The advantage of the approach developed in this paper is that users can develop their own schedulers for multi-core parallelism, taking advantage of the most efficient low-level asymmetric messaging, without having to cope with distributed debugging. In this section we outline the algorithm for the scheduler we are using to test this framework.

This scheduling problem is NP-complete, and finding good approximation algorithms is a current research problem, especially for the (harder) case of multi-processor scheduling [21]. Devising a good heuristic algorithm depends on understanding the properties of a good solution. Just as in the single-pipelined-processor ILP problem, the most important feature of a good schedule is the separation of loads and instructions consuming the results. We use a greedy algorithm which seeks to move the signal/data transfers as far from the computations as possible.

We divide the process into three stages, guided by the code graph transformation sequence shown in Fig. 3. These graphs present a multi-core implementation of matrix multiplication, specialised, for the sake of readability, for multiplying a matrix A tiled into 1×1 blocks with a matrix B tiled into 1×2 blocks, splitting the two resulting block multiplications, labelled "$*$", over two cores 0 and 1.

These edges labelled with "$*$" should be considered as nested edges, labelled with a control-flow graph representing the loop structure of a software-pipelined submatrix multiplication program; the loop body in that control-flow graph is labelled with a pure data-flow graph assembled from several stages of an unrolled submatrix multiplication loop body [4].

The three stages, described in more detail in separate subsections below, essentially produce a schedule for the final, concurrent data flow graph by approximating it first with a schedule for the data-flow graph and then with one for the intermediate, "distributed data-flow" graph:

Payload scheduling: Enumerate the data blocks with sources, in the order in which they are used in computations (as inputs or outputs of pure functions, or as sources for transfers to main node memory), and enumerate the computations in the order they will be executed (this can be done once for each core or in a unified time-line).

We consider this as scheduling a pure data-flow graph, which is shown (with cores assigned) for a trivial 2-core example in Fig. 3 (left).

Payload buffer allocation and communication generation: Generate ordered lists of instructions for each core from the initial lists, converting pure functions into RunComputations, and inserting data transfers and signalling as required to get data blocks into local buffers before pure computations.

We consider this as refining the previous schedule for the corresponding distributed data-flow graph, i.e., with explicit "mv" edges representing data transfer between cores, as shown for the same example in Fig. 3 (middle), and simultaneously performing resource allocation (corresponding to register allocation in simple data-flow graphs).

Communication Scheduling: Reorder the instruction lists to better hide latency of signals.

Since the "mv" edges of the previous graphs are implemented using sequences of instructions with widely varying latencies and limited dependencies, these instructions must be scheduled separately, which amounts to producing a schedule for the concurrent data-flow graph shown
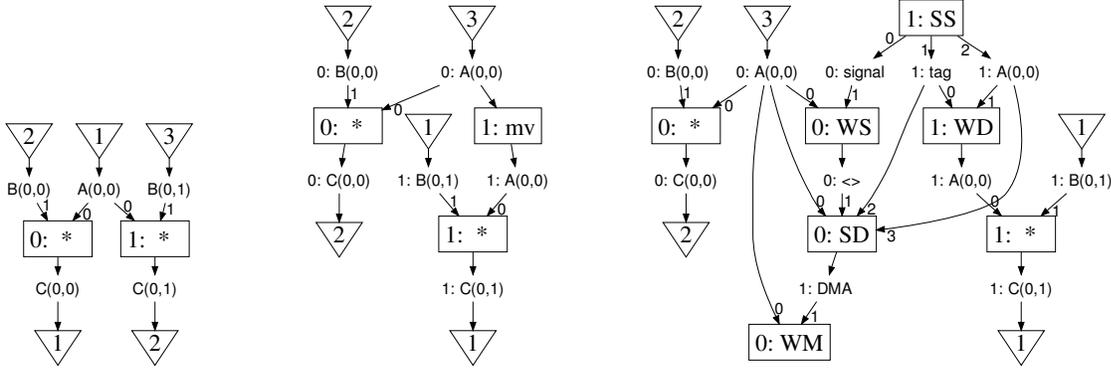
Figure 3: Pure, distributed, and concurrent data-flow graphs

(again for the same example) in Fig. 3 (right). For that graph, note that the WaitDMA instruction "WM" does not produce a (data) output, but still needs to be scheduled since it is strict like all other instructions here. It also does not have a control output since nothing directly depends on its completion; its only effect is to keep the buffer allocation for the buffer containing A(0,0) on core 0 alive until after its completion: the outgoing control edge added to "WM" by scheduling it will then together with a correct allocation ensure that there are no writes to that buffer before completion of that "WM".

6.1   Payload Scheduling

This first stage is separated out because the key to efficiently utilising resources in high efficiency multi-core architectures is the maximisation of data reuse within the cores, and hiding of latency.

Access bandwidth to main node memory (and to distant memory on other nodes) is bounded by hardware interfaces which cannot grow at the rate computational performance is growing. The first stage orders computations so that input and intermediate data can be maximally reused. In the future, on multi-core processors with more processors and topologically non-trivial on-chip networks, this is the point at which operations can be reordered to distribute inter-core communication sufficiently over fabric bottlenecks. On the Cell BE, main node memory traffic is the bottleneck for all of our target computations. We will refer to the list of buffer uses as *consumers*.

6.2   Payload Buffer Allocation+Communication Generation

The second stage maximises latency hiding by filling buffers with data blocks required for future computations as soon as buffers become available. This is done by iterating the following six stages:

1. Look for unoccupied buffers on each core and where found assign consumers, generating instructions to send and receive signals and initiate DMA transfers, except for output blocks, which require a buffer reservation, but no action.

2. Age all pending transfers by the estimated time to initiate transfers.

3. Look at the next instructions on each core, and if all of the required blocks have been assigned to local buffers, and inputs are available, then generate the RunComputation or StoreData instruction.

4. Adjust use counts on data blocks following computation.

5. Age all pending transfers by the estimated computation time for executed computations.

6. Find the oldest pending data transfer; add a WaitData instruction to the instruction stream for its core, and mark the data block in the target buffer as available.

6.3   Communication Scheduling

The resulting schedule makes maximal use of the available buffers to hide the latency of data transfers. It does not hide the latency of signals. If signals are implemented has high-priority, low-bandwidth side-band communication, it would not be necessary to worry about this. If signals share a single data bus, signal latency may be as long as data latency. On the Cell BE the situation is somewhere in between, as is likely to be the case for most future architectures. Signals are transmitted on the same bus as data, but the "bus" is implemented as multiple token rings running in different directions. So shorter signals running in the opposite direction to data will tend to have significantly shorter latency. Nevertheless, it is important to be able to hide this latency as well. We do this in a final stage, using what is akin to peephole optimisation, by locating the matching WaitSignal and SendData pair and moving the WaitSignal forward across a number of instructions representing the smallest amount of computation above the minimum of a fixed expected latency, and a fraction of the total computation time between the WaitSignal and SendData in the input instruction stream.

7   Concurrency Verification

We envisage two verification steps which make sense in the context of multi-core parallelism: Reduction of the parallel program to a pure data flow graph, and matching the data flow graph to a specification. For the types of problems we are immediately interested in, like structured linear algebra, it is easy to generate single- and multi-level pure data flow graphs and convert them to an appropriate normal form. This makes the second step easy, and the first step the more interesting.

The first step requires the verification that the introduction of multi-core parallelism does not modify the semantics of a program. Even where normal forms for pure data flow graphs are not available, this means that testing can be as efficient as it is for single-threaded programs. This is similar to the graph transformation strategy we are applying successfully to the development of code transformations for VIRP.

In practice, incorrect programs are as interesting to the programmer whose job it is to fix them as are correct programs. On failure, our verifier will identify a program location and type of error. This is much easier to do with an ordered input program than with a set of communicating processes.

This brings us back to the superscalar, out-of-order model. Although compilers know that instructions are executed out of order, they have to be presented in a linear order, and the semantics must match the sequential program semantics. This is the case for verified programs presented in our DSL. The difference from the superscalar CPU case is that some programs cannot be verified, and therefore do not have the same semantics as the equivalent sequential programs.

We will use the following terminology:

- A *program* is a list of pairs (*CoreId, Instruction*).

    We assume that the instructions originate in this order and are transmitted to cores in this order. Since each core buffers its instructions and executes them without additional synchronisation, the original program order is not necessarily the same as the resulting execution order.

However, due to limited buffer sizes there is a limit to the distance within the list between any two instructions whose execution overlaps in time. Therefore, considering a single list is more restrictive than a separate list of instructions for each core.

The situation here corresponds to the dispatch queues and instruction fetch buffers on superscalar CPUs, which result in a maximum number of instructions in-flight[5].

We do not use the maximum in-flight property, but do use the presentation order, and we will add restrictions to the presentation order below.

- A program is *order independent* if given the same input (in main memory input locations) all possible execution orders terminate and produce the same output (in main memory output locations), although intermediate values computed and temporarily stored in private or global memory may differ.

- A program is *locally sequential* if every $(c_1, \mathsf{SendSignal}\, c_2\, i)$ is followed by a corresponding $(c_2, \mathsf{WaitSignal}\, i)$, and every $(c_1, \mathsf{SendData}\, b\, c_2\, t\, m)$ is followed by corresponding $(c_2, \mathsf{WaitData}\, t)$ and $(c_1, \mathsf{WaitDMA}\, m)$ instructions.

   Note that it is easy to construct order independent programs without this property, simply by ordering the instructions by core.

- A program is *safely sequential* if it is locally sequential and order independent.

The programmer wants to know programs are order independent. Checking this is very expensive. The programmer understands sequential programs, and with some difficulty can understand parallel programs "close to" sequential programs. Weakly sequential programs are our attempt to formalise this concept, and we may restrict it to improve programmer understanding in the future as we gain experience writing and generating a wider range of programs. The key to usability, however, is the fact that we can efficiently identify the safely sequential programs within the class of locally sequential programs.

To summarize these definitions using relational language: instructions on a single core are totally ordered; instructions on different cores have a partial order derived from the use of synchronizing primitives; the partial order is the meet of all possible execution orders; the presentation order (order in the list) is a total order, and we will give conditions that the imply that partial order is a sub-relation thereof, and in particular, that the presentation order is a valid execution order.

One reason we can identify the safe subset efficiently, is that we can identify what the "intended" result should be by using the pairing of send and receive instructions, and from there we can verify that these are the only results by keeping track of a bounded set of state information (with the size of the information being $\mathcal{O}(\mathrm{cores}^2)$).

7.1   Motivating Example

Before explaining the efficient method of verification, it is important to understand that locally sequential programs are not necessarily sound.

For simplicity, the following locally sequential program uses signals, but the same issues arise with data transmission:

---

[5]The instructions *in-flight* at a given time are all of the instructions in the process of being decoded by the CPU, dispatched to execution units, executing (pipelined or non-pipelined), or being completed (having their results recorded in register files, store queues).

index	core 1	core 2	core 3
1		long computation	
2	SendSignal $s \to c2$		
3		WaitSignal s	
4		computation	
5			SendSignal $s \to c2$
6		WaitSignal s	

Remember that each core executes independently of the other cores, except where explicit wait instructions block execution until some kind of communication (signal, change in data tag, DMA) is confirmed to have completed. Therefore, in this case the most likely instruction completion order has core 3 executing the SendSignal as soon as it is queued, allowing the signal to be sent before core 2 has received the core 1's signal and cleared the signal hardware:

index	core 1	core 2	core 3
2	SendSignal $s \to c2$		
5			SendSignal $s \to c2$
	second signal overlaps the first, only one registered		
1		long computation	
3		WaitSignal s	
4		computation	
	no signal is sent, so the next WaitSignal *blocks*		
6		WaitSignal s	

To be precise, completion of the SendSignal means that the signal has been initiated by the sender, and reception may be delayed, so the signal from core 3 could even arrive before the signal from core 1. In either case, neither signal will arrive after the first WaitSignal, so the second WaitSignal will wait forever, and this program execution will not terminate.

The problem is caused because there are no signals or data transmissions enforcing completion of instruction 5 to follow completion of instruction 3.

This example, when considered as part of a longer program, also demonstrates a possible safety violation with the valid completion order:

index	core 1	core 2	core 3
1		long computation	
5			SendSignal $s \to c2$
3		WaitSignal s	
4		computation **using wrong assumptions**	
2	SendSignal $s \to c2$		
6		WaitSignal s	

In this case, the computation 4 might rely on assumptions that are only available once the SendSignal 2 completes — for example, it might initiate a DMA to core 1, which could arrive before some earlier instructions on core 1 are still writing to the DMA's target area, thus invalidating the DMA transfer.

7.2   Strictly forward inspection of partial order

We have seen that locally sequential programs are not always sound. The advantage that locally sequential programs have over general programs is that order inducing instructions must occur before instructions whose well-definedness depends on them. Using simple inference to maintain a list of known ordering relations between instructions on different cores, we can inspect the program in order (even though it can execute out of order).

Good Program

index	core 1	core 2	core 3
1		long computation	
2	SendSignal $s \to c2$		
3		WaitSignal s	
4		computation	
5		SendSignal $s \to c1$	
6	WaitSignal s		
7	SendSignal $s \to c3$		
8			WaitSignal s
9			SendSignal $s \to c2$
10		WaitSignal s	

Figure 4: Weakly sequential program

Referring to Fig. 4, we see that after instruction 3, we know that instructions on core 2 will complete after instruction 2 on core 1 (and any previous instructions). We know nothing about the relative execution of instructions other than this until instruction 6. At this point we know that instructions after and including 6 on core 1 must execute after any instruction before instruction 5 on core 2. After instruction 8, we have the relation that this and further instructions on core 3 execute after instructions up to instruction 7 on core 1, but since instruction 7 on core 1 executes after instruction 5 on core 2, we also know that instruction 8 on core 3 executes after instruction 5 on core 2.

Contrast this with the example in Sect. 7.1, where we never have any relations between instructions on core 3 and other cores. Keeping track of these relations is the key to efficient verification of correctness.

There is one other class of (bad) example to keep in mind: the situation where two cores send the same signal to a third core without a wait on the third core to separate them. This case is visually apparent, because the sends overlap in program order, but we have to safeguard against it nonetheless.

Clearly, the same considerations apply to data transfers. With data transfers, we must additionally check the values of the data tags, because if the existing tag matches the tag being transferred in, the WaitData instruction will never block execution, even if the data is not available.

7.3   State

For brevity, we will now write I instead of *Instruction* for the set of instructions, C instead of *CoreId*, S instead of *SignalId*, and L instead of *LocalAddress*. Let D be the set of DMA Tags. Let T be the set of data tags.

In the following, we assume a fixed program P. Depending on this program P, let $N = \{0, \ldots, \text{length}(P) - 1\} \subseteq \mathbb{N}$ be the set of indices into P considered as a list, so that we can now consider the program as a total function $P : N \to C \times I$ instead. N is considered to be ordered with the standard ordering of the natural numbers.

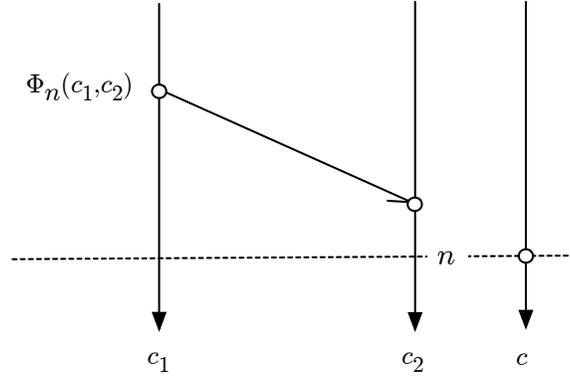For each core $c : C$, define $N_c = \{n : N \mid P(n)_1 = c\}$ to be the corresponding restriction of the index

Figure 5: For convenience, $\Phi_n(c_1, c_2)$ is defined for all n, even if $n \notin N_{c_2}$, to be the index of the last instruction on c_1 known to complete before an instruction on c_2 up to and including index n.

set to instructions for c.

We are now interested in the partial strict-ordering $\prec$ (depending of course on P) on N where $n_1 \prec n_2$ means "the instruction (of P) at n_1 *necessarily* completes before the instruction at n_2". The reflexive closure of this is frequently known as "happens-before relation" [13], or "Mazurkiewicz's trace" [22, 15].

For a fixed core, $c : C$, the instructions in the program assigned to c are completed in order, so we demand that, for each core $c : C$, the restriction of $<$ to N_c is included in $\prec$:

$$\forall c : C \ . \ \forall n_1, n_2 : N_c \ . \ n_1 < n_2 \Rightarrow n_1 \prec n_2 \tag{1}$$

There is no a priori order between instructions assigned to distinct cores, and may lead to unsound programs like the example in Sect. 7.1. Ordering is determined by transition dependencies, which in our case means blocking communication.

It would be sufficient to calculate $\prec$ completely, but only a small portion of this partial order is required at a time. Since instructions on a single core are totally ordered, it is easy to calculate the entire relation $\prec$ if, for each instruction and distant core, the latest instruction known to complete before and the earliest instruction known to complete after this instruction are stored.

We will see that it is sufficient to store only the latest instruction known to complete before the current instruction, and convenient to repeat information for all cores, in the form of a "Φollows" map indexed by $n \in N$:

$$\Phi_n : C \times C \to N \tag{2}$$
$$\Phi_n(c_1, c_2) = \max_{\leq}\{m : N_{c_1} \ | \ \exists n' : N_{c_2} \ . \ n' \leq n \Rightarrow m \prec n'\} \ .$$

Note, in particular, that if $n \notin N_{c_2}$ this definition does not say that is is the latest instruction on c_1 such that all future instructions on c_2 follow m. This property could be used instead, but is more expensive to compute since it requires information about future instructions. Fig. 5 shows that the instruction at index $\Phi_n(c_1, c_2)$ is always a send instruction on core c_1.

For the verification method, we need to define a function

$$\widetilde{\bigvee} : (C \times N \times F) \times (C \times F) \to F \ ,$$

where $F = C \times C \to N$ models a slice of Φ. $\widetilde{\bigvee}$ combines the slices of the partial order at a corresponding pair of send/wait instructions that induce an order on the completion of the wait and instructions on other cores. This potentially strengthens what we know about $\Phi_{n_2}(c, c_{\text{Wait}})$ for all c, *i.e.* about the

last instructions known to complete before the wait and future instructions on the destination core.

$$\widetilde{\bigvee}((c_1, n_1, \Phi_{n_1}), (c_2, \Phi_{n_2}))(c', c'') =$$

$$\begin{cases} max\{\Phi_{n_2}(c', c''), n_1\} & c' = c_1, c'' = c_2 \\ max\{\Phi_{n_2}(c', c''), \Phi_{n_1}(c', c_1)\} & c'' = c_2 \\ \Phi_{n_2}(c', c'') & \text{otherwise.} \end{cases} \tag{3}$$

Our method of verification is a construction of the following functions inductively in the instruction index, n.

$$\sigma_n : C \times S \to \begin{pmatrix} C \times N \times F \\ \oplus \\ N \end{pmatrix} \tag{4}$$

$$\Delta_n : C \times L \to \begin{pmatrix} T \times C \times N \times F \\ \oplus \\ T \times N \\ \oplus \\ T \times 2^D \\ \oplus \\ T \times D \end{pmatrix} \tag{5}$$

$$\Phi_n : C \times C \to N \tag{6}$$

where $\oplus$ means disjoint union, and 2^D is the power set of D.

To calculate X_{n+1}, where $X_n = (\sigma_n, \Delta_n, \Phi_n)$, it is only necessary to have access to X_n, because we make copies of slices of Φ which are needed in future steps.

The disjoint unions can be understood in the following way: for σ's range, there are two cases for signal status. Case I: a signal is expected from a core $c : C$, at instruction $n : N$, with $\Phi' : F$ being the known partial order at n. Case II: the signal has been received at instruction $n : N$.

For Δ, the cases are that

1. an incoming transfer with tag in T, from core in C started at instruction in N with known partial order in F is expected;

2. the data tag is in T, and the data was last used at an instruction in N;

3. the data buffer is the source of one or more DMA transfers whose list of (completion) tags are in 2^D, and the data tag is in T;

4. a incoming DMA with tag in D was started, the data tag is in T.

Initially, σ_0 maps every argument to -1 (indicating that the last signal was received before the beginning of the program), and Δ_0 maps to the pair $(\langle unused \ data \ tag \rangle, -1)$. Φ_0 maps every argument to -1 as well, indicating that nothing is known about ordering.

The maps σ_{n+1}, Δ_{n+1} and Φ_{n+1} are constructed from the maps at n and the instruction at n executing on core c using the universal rule $\Phi_{n+1}(c, c) = (n)$, and specific rules for the following cases:

SendSignal c' s'

      Divide into two cases according to the form of $\sigma_n(c', s')$.

If $\sigma_n(c', s') = (c'', n', \Phi')$, this is an error, because this signal may be sent by both c' and c''.

If $\Phi_n(c', c) \geq \sigma_n(c', s')$, then we know that the last instruction to consume the signal s' on c' at $\sigma_n(c', s')$ completes before $\Phi_n(c', c)$ on c' which (by definition) occurs before the current instruction on c. Let $\Phi_{n+1} := \Phi_n$, $\Delta_{n+1} := \Delta_n$, and $\sigma_{n+1}(c', s') := (c, n, \Phi_n)$, $\sigma_{n+1}(c'', s'') := \sigma_n(c'', s'')$ for other values of (c'', s'').

If the inequality is not satisfied, the program is not sound, because we do not have $n' \prec n$, meaning that this signal could arrive before the previous WaitSignal s' on c', which is the case of the bad example program.

WaitSignal s'

If $\sigma_n(c, s') = (c'', n', \Phi')$, then we know that the signal s' is coming, or will come from core c'', and we can use the fact that the wait completes after the send, to update the follows map. Let $\Phi_{n+1} := \widetilde{\vee}((c'', n', \Phi'), (c, \Phi_n))$, $\Delta_{n+1} := \Delta_n$, $\sigma_{n+1}(c, s') := (n)$ otherwise $\sigma_{n+1}(c''', s''') := \sigma_n(c''', s''')$.

If $\sigma_n(c, s') = (n')$, this is an error, or the program is not locally sequential, as required for this analysis.

SendData l' λ c'' l'' t' d

At the Source: If $\Delta_n(c, l') = (t'', n'')$ then the no DMAs are pending for this buffer, so record the send $\Delta_{n+1}(c, l') := (t'', \{d\})$.

If $\Delta_n(c, l') = (\tilde{t}, d')$, then an incoming DMA is progress, and it is not safe to start an out-going DMA, so this is an error.

If $\Delta_n(c, l') = (\tilde{t}, D')$ for some subset $D' \subset D$, the there are already out-going DMAs in progress. If $d \in D'$, we are illegally reusing the DMA tag. If $\tilde{t} \neq t''$, then two different tags are being used for outgoing DMAs, which is not supported (because it would result in an unknown data tag being send for the previously initiated transfer). Otherwise, record its' use by $\Delta_{n+1}(c, l') := (t, D' \cup \{d\})$, and copy the other maps from n to $n + 1$.

Otherwise, this buffer is the target of a DMA from another core, which is an error.

At the Destination: If $\Delta_n(c'', l'') = (t'', n'')$, $t'' \neq t'$, and $n'' \leq \Phi_n(c'', c)$, then the destination buffer has no pending DMAs after instruction n'', the new tag does not matches the old tag (so it will be detectable by the WaitData), and the current instruction is known to complete ($\Phi_n(c'', c)$) after the last instruction (n'') to have used the target buffer, so it is safe, and we set $\Delta_n(c'', l'') := (t', c, n, \Phi_n)$. Otherwise report the error that multiple incompatible DMAs with source and target (c'', l'') were detected, or the data tag is illegally reused.

WaitData l' t'

If $\Delta_n(c, l') = (t', c'', n', \Phi')$, then we are waiting for this data, and have stored the follows map at the send instruction which we can meet with the map at the current instruction: $\Phi_{n+1} := \widetilde{\vee}((c'', n', \Phi'), (c, \Phi_n))$. Let $\sigma_{n+1} := \sigma_n$, and $\Delta_{n+1} := \Delta_n$ except $\Delta_{n+1}(c, l') := (t', n)$, which indicates that this buffer is safe to use after the current instruction.

Otherwise no incoming DMA from another core preceded this instruction, so the program is not locally sequential, or this DMA overlaps DMAs initiated locally, which is unsafe.

LoadMemory l' λ g' d

Check that d does not appear in the image of $\Delta_n(\{c\} \times L)$, otherwise the uses of this tag overlap, which is an error.

If $\Delta_n(c, l') = (t, n')$, the the buffer has no pending IO, and we can initiate an incoming DMA by setting $\Delta_{n+1} := \Delta_n$ except $\Delta_{n+1}(c, l') := (t, d)$, $\sigma_{n+1} := \sigma_n$ and $\Phi_{n+1} := \Phi_n$.

Otherwise the target buffer is the source or target for transfer which may not have completed, and this will produce an indeterminate result.

StoreMemory l' λ g' d

Check that d does not appear in the image of $\Delta_n(\{c\} \times L)$, otherwise the uses of this tag overlap, which is an error.

If $\Delta_n(c, l') = (t, n')$, then no other IO is pending, and the store is safe. Set $\Delta_{n+1} := \Delta_n$ except $\Delta_{n+1}(c, l') := (t, \{d\})$ $\sigma_{n+1} := \sigma_n$ and $\Phi_{n+1} := \Phi_n$.

If $\Delta_n(c, l') = (t, D')$, other out-going DMAs are pending, which is allowed. So set $\Delta_{n+1} := \Delta_n$ except $\Delta_{n+1}(c, l') := (t, D' \cup \{d\})$ (to record the new DMA tag we are waiting for), $\sigma_{n+1} := \sigma_n$ and $\Phi_{n+1} := \Phi_n$.

Otherwise the source buffer is the target for a pending transfer, and this will produce an indeterminate result.

WaitDMA d

The DMA tag could be associated with incoming or outgoing IO. In either case $\exists l$ such that $\Delta_n(c, l) = (t, d)$ (incoming), or $\Delta_n(c, l) = (t, D')$ and $d \in D'$ (outgoing), and l is unique. Otherwise, there is an error, because no DMAs are pending using this tag. This either indicates an unsound program or a non-locally sequential presentation.

If $\Delta_n(c, l') = (t', \{d\})$ then $\Delta_{n+1}(c, l') := (t', n)$.

If $\Delta_n(c, l') = (t, D')$ then $\Delta_{n+1}(c, l') := (t, D' \setminus \{d\})$

If $\Delta_n(c, l') = (t', d)$ then $\Delta_{n+1}(c, l') := (t', n)$.

For other values $\Delta_{n+1} := \Delta_n$. Also, $\sigma_{n+1} := \sigma_n$ and $\Phi_{n+1} := \Phi_n$.

RunComputation x $(l', l'', ...)$ $(\tilde{l}', \tilde{l}'', ...)$ $(p', p'', ...)$

Inputs: If for every $l \in \{l', l'', ...\}$, $\Delta_n(c, l) = (t, n')$ or (t, D') then $\Delta_{n+1}(c, l) := (t, n)$ or (t, D') (respectively), otherwise, the buffer has pending incoming IO, which makes this computation unsound.

Outputs: If for every $l \in \{l', l'', ...\}$, $\Delta_n(c, l) = (t, n')$ then $\Delta_{n+1}(c, l) := (t, n)$, otherwise, there is pending IO and this modification makes the program unsound.

Other instructions have no effect, so $\Phi_{n+1} := \Phi_n$, $\Delta_{n+1} := \Delta_n$, and $\sigma_{n+1} := \sigma_n$.

Theorem 7.1 *A locally sequential program is order independent* iff *the inductive verification described in this section terminates without error.*

For the proof of this theorem, we need the following lemmas:

Lemma 7.2 *Each $d \in D$ occurs at most once in $\Delta_n(\{c\} \times L)$.*

Proof: This property is preserved because the verifier starts with no elements of D in the images of any of the maps, and all rules check for previous use of a tag before constructing the $n+1$ maps from the n maps and signal an error if multiple occurrences of some d would result.

Lemma 7.3 *In a locally sequential program which passes the verification procedure, the only signal a $(n, c, \mathsf{WaitSignal}\ s)$ can trap is the one in $\sigma_n(c, s)$.*

Proof: Assume that there is another signal which can be trapped. Since the program is locally sequential, the "matched" SendSignal c s must precede the WaitSignal s in presentation order. There are two cases to check: the other send precedes the wait, or follows the wait.

1. If it precedes the wait, we have two different SendSignal c s instructions sending the same
   signal to the same core, $c \in C$, preceding the WaitSignal, with unrelated instructions potentially
   interleaved. However this situation would not pass the verification, because at the second send,
   $(n, \mathsf{SendSignal}\ c\ s)$, $\sigma_n(c, s) = (c', n', \Phi')$ where n', c' are the index and location of the first send,
   which would cause the verifier to return an error.

2. If the second SendSignal instruction follows the wait in presentation order, but both sends can
   execute before the wait, then at the second send, the verifier will find $\Phi_n(c', c) < n'$, where
   (n, c) are the coordinates of the second send, and (n', c') of the intermediate wait. This would
   have caused the verifier to signal an error, precisely because it could not guarantee the safety of
   the second send.   □

Lemma 7.4 *In a locally sequential program which passes the verification procedure, the only data a*
$(n, c, \mathsf{WaitData}\ l\ t)$ can observe arriving is the one in $\Delta_n(c, l)$.

Proof: Analogous to the proof of Lemma 7.3.   □

Lemma 7.5 *The presentation order of a locally sequential program is a valid execution order up to n*
if verification produces no error up to instruction n.

Proof: Assume the contrary, and let m be the first instruction which is not executable after instruc-
tion $m - 1$ and all previous instructions have completed. The only instructions which may not be
executable are the wait instructions (for a signal, for a data tag, or for a DMA completion tag). Since
m is the first non-executable instruction, all previous instructions have completed, which includes the
corresponding send instruction (since the program is presentation is locally sequential). So m is also
executable, which is the contradiction.   □

Lemma 7.6 *For all $c', c \in C$, $n \in N_c$, $n' \in N_{c'}$, if $n > \Phi_{n'}(c, c')$, then there exists an execution*
order ϵ in which $n' <_\epsilon n$.

Proof: Construct an execution order ϵ as follows. Let $U_{c''} = \{m'' \in N_{c''} : m'' \leq \Phi_{n'}(c'', c')\}$, and
$\mathcal{U} = \bigcup_{c'' \in C} U_{c''}$. A modification of the proof of Lemma 7.5 shows that the presentation order is a
valid execution order for the subset $\mathcal{U}$. The only new point is that the instructions not in $\mathcal{U}$ are not
required to make the instructions of $\mathcal{U}$ executable, because the follows maps capture the supremum
of the necessary instructions for the subset $\mathcal{U}$. Since n is not in the execution order up to this point,
$n > n'$ in this order, as required.   □

7.4   Proof of Theorem

To prove that a locally sequential program which is order-independent will pass verification we consider
the converse situation, a locally sequential in which an error is flagged. For each case of such an error,
one must construct a possible execution order which results in a deadlock. We do this for the hardest
case, the errors raised during the processing of a SendSignal. The other errors can be treated in the
same way, or they obviously indicate a failure to be locally sequential (waits preceding sends).

$(n,\ c,\ \mathsf{SendSignal}\ c'\ s)$

(1) Error caused by $\sigma_n(c', s) = (c'', n', \Phi')$ where we have a signal s which could be sent by
both c and c'' to c'. Presentation order is a valid execution order up to n if it produces no
error up to instruction n by Lemma 7.5. Therefore both signals could arrive at c' before the

next WaitSignal, in which case both would be consumed. This would leave a second WaitSignal without a paired SendSignal according to the 1-1 pairing of SendSignal and WaitSignal required in a locally sequential program, and a deadlock would result. (Note that even if the program is malformed in other ways, as long as the sends and waits are paired, each send can make at most one wait executable, so the pairing is the only required property to insure a deadlock, unless some other problem manifests itself first.)

(2) Error caused by $\sigma_n(c', s') > \Phi_n(c', c)$. This tells us that the last instruction to consume signal s' on c' at $\sigma_n(c', s')$ does not necessarily complete before $\Phi_n(c', c)$ on c'. By Lemma 7.6, there is an execution order in which the signal sent at n completes before the wait at $\sigma_n(c', s')$, in which case the wait would capture two signals, and there would be an unpaired wait later in the program to cause a deadlock, as above.

To prove the converse, let P be a locally sequential program passing verification. We must first show that the program has no deadlocks, and always produces the same result.

Assume that there is an execution order with a blocking instruction. Let n be the first instruction in presentation order which blocks. In analogy with the proof of Lemma 7.5, this instruction must be a wait instruction, which by Lemma 7.3 and Lemma 7.4 is paired with a unique send instruction which precedes it in the presentation order. Since that instruction did not deadlock (because n is the first such instruction), it is executable, making n executable, implying that a verified program can have no deadlocks.

To show that the (non-deadlocking) result is order independent, we can symbolically execute it, checking at each step that the results do not depend on execution order. To do this, modify the image of Δ_n to include an expression type, E, with different meanings for each case in the disjoint union:

$T \times C \times N \times F \times E$ an incoming transfer is expected, and when it arrives, E will be the contents of the buffer;

$T \times N \times E$ the buffer contains the expression E, which was last used at instruction $n \in N$;

$T \times 2^D \times E$ the buffer contains the expression E, and is the source for outgoing transfers;

$T \times D \times E$ the buffer is the target of an incoming transfer from memory, and when it completes the buffer will contain E.

New expressions are formed by a RunComputation at instruction n from the expressions in the input buffers in Δ_n and the new expressions are stored in Δ_{n+1}. The verification mechanism insures that the source and destination buffers in Δ are in the definite state $T \times N \times E$. Sourcing buffers in other (indefinite) states would have been flagged as an error. Input expressions are formed by memory loads, and an additional map from memory store locations to expression values represents the output of the program. Memory outside the on-core buffers is considered to be static input, or write-once output data. This concludes the proof of the theorem.   □

For the second step in verification, one must compare the expressions generated in this way with specifications for the program. For computations of limited size, this can be done by using a nodes in a DAG with sharing as the type E. The result is a term forest, which can easily be compared to a specification in the same form. This procedure would be considerably more complicated for programs whose corresponding DAGs would not fit in program memory, but it is the nature of this approach that computation, and specifications are nested, so that the size of DAGs at each level are manageable. So far, this has always been the case.

8   Conclusion

We have shown that the control flow required to efficiently utilise multiple light-weight cores can be abstracted out of the program into a separate language, which we use to label the middle layer of Coconut nested code graphs.

We have drawn an analogy between this language and a RISC instruction set used in a superscalar processor. This analogy leads naturally to the idea that the multi-core level of the graph be presented in order, for out-of-order execution. With this presentation, it was possible to verify the correctness of this level of parallelism in an efficient manner.

In the future, we will extend this language to handle nested parallelism of this type (including multiple nodes each containing multiple cores). We will also address multi-phase computations, including the changing use of memory, loading and unloading of computations, and the overlapping of fill/drain between phases.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, G. Vidal. *Operational Semantics for Declarative Multi-Paradigm Languages*. Journal of Symbolic Computation **40**(1) 795–829, 2005. 2.1

[2] C. K. Anand, W. Kahl. *Code Graph Transformations for Verifiable Generation of SIMD-Parallel Assembly Code*. In A. Schürr, M. Nagl, A. Zündorf, eds., Applications of Graph Transformations with Industrial Relevance, Third Intl. Symp., AGTIVE 2007, Participants' Proceedings, pp. 213–228, 2007. 2, 2.1

[3] C. K. Anand, W. Kahl. *A Domain-Specific Language for the Generation of Optimized SIMD-Parallel Assembly Code*. SQRL Report 43, McMaster University, 2007. available from http://sqrl.mcmaster.ca/sqrl_reports.html. 2.1, 3

[4] C. K. Anand, W. Kahl. *MultiLoop: Efficient Software Pipelining for Modern Hardware*. In: CASCON '07: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research, pp. 260–263, New York, NY, USA, 2007. ACM. 2.2, 6

[5] P. Bellens, J. M. Perez, R. M. Badia, J. Labarta. *CellSs: a Programming Model for the Cell BE Architecture*. IEEE , 2006. 1

[6] R. Bruni, J. Meseguer, U. Montanari, V. Sassone. *Functorial Models for Petri Nets*. Information and Computation **170**(2) 207–236, 2001. 2.3, 2.4

[7] I. B. M. Corporation, S. C. E. Incorporated, T. Corporation. *Cell Broadband Engine Programming Handbook*. IBM Systems and Technology Group, Hopewell Junction, NY, 1.0 edition. 3

[8] A. Corradini, F. Gadducci. *An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories*. Applied Categorical Structures **7**(4) 299–331, 1999. 1, 2, 2.1, 2.4

[9] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. *Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach*. In G. Rozenberg, ed., Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations, Chapt. 3, pp. 163–245. World Scientific, Singapore, 1997. 2

[10] A. Corradini, F. Gadducci, W. Kahl. *Term Graph Syntax for Multi-Algebras*. Technical Report TR-00-04, Dipartimento di Informatica, Università di Pisa, 2000. 2, 2.1

[11] A. Corradini, F. Gadducci, W. Kahl, B. König. *Inequational Deduction as Term Graph Rewriting*. In D. Plump, ed., TERMGRAPH 2002, International Workshop on Term Graph Rewriting, Electronic Notes in Computer Science **72**, pp. 31–44. Elsevier Science B.V., 2007. 2.1

[12] E. A. Emerson. *Temporal and Modal Logic*. In J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics, pp. 995–1072. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990. 2.3

[13] C. Flanagan, P. Godefroid. *Dynamic Partial-Order Reduction for Model Checking Software.* In: Principles of Programming Languages, POPL 2005, pp. 110–121. ACM, ACM, 2005. 1, 7.3

[14] F. Gadducci, U. Montanari. *Axioms for Contextual Net Processes.* In K. Larsen et al., eds., ICALP '98, LNCS **1443**, pp. 296–308. Springer, 1998. 2.3, 2.4

[15] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Vol. 1032. Springer-Verlag Inc., New York, NY, USA, 1996. 7.3

[16] G. Goumas, M. Athanasaki, N. Koziris. *Automatic code generation for executing tiled nested loops onto parallel architectures.* In: SAC '02: Proceedings of the 2002 ACM symposium on Applied computing, pp. 876–881, New York, NY, USA, 2002. ACM. 1

[17] G. Goumas, N. Drosinos, M. Athanasaki, N. Koziris. *Automatic parallel code generation for tiled nested loops.* In: SAC '04: Proceedings of the 2004 ACM symposium on Applied computing, pp. 1412–1419, New York, NY, USA, 2004. ACM. 1

[18] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki. *Synergistic Processing in Cell's Multicore Architecture.* IEEE Micro **26**(2) 10–24, 2006. 4

[19] B. Hoffmann, D. Plump. *Jungle Evaluation for Efficient Term Rewriting.* In J. Gabrowski, P. Lescanne, W. Wechler, eds., Algebraic and Logic Programming, ALP '88, Mathematical Research **49**, pp. 191–203. Akademie-Verlag, 1988. 2.1

[20] W. Kahl, C. K. Anand, J. Carette. *Control-Flow Semantics for Assembly-Level Data-Flow Graphs.* In W. McCaull et al., eds., 8th Intl. Sem. Relational Methods in Computer Science, LNCS **3929**, pp. 147–160. Springer, 2006. 2, 2.1

[21] S. Leonardi, D. Raz. *Approximating total flow time on parallel machines.* J. Comput. Syst. Sci. **73**(6) 875–891, 2007. 6

[22] A. Mazurkiewicz. *Trace Theory.* In: Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, LNCS **255**, pp. 279–324. Springer, 1986. 7.3

[23] P. S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1997. 1, 3

[24] D. Plump. *Term Graph Rewriting.* In H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, eds., Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, Chapt. 1, pp. 3–61. World Scientific, Singapore, 1999. 2.1

[25] V. Sassone. *On the Algebraic Structure of Petri Nets.* Bull. European Assoc. Theoretical Computer Science **72** 133–148, 2000. 2.3, 2.4

[26] M. Sleep, M. Plasmeijer, M. van Eekelen, eds. *Term Graph Rewriting: Theory and Practice.* Wiley, 1993. 2

[27] M. Snir, S. W. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. *MPI — The Complete Reference.* Scientific and Engineering Computation Series. MIT Press, 1996. 1, 3

[28] Gheorghe Ştefănescu. *Network Algebra.* Springer, London, 2000. 1